Instituto Superior Técnico
Mestrado em Engenharia Electrotécnica e de Computadores

Parallel and Distributed Computing

Project Report - MPI

Gonçalo Bernardino Frazão
Gustavo Rodrigues Vítor
João Paiva de Castelo-Branco

Prof. José Monteiro

2023/2024

1 - Introduction

This report covers the third part of the Parallel and Distributed Computing project for the academic year 2023/2024. The project involves creating a 3D version of Conway's Game of Life using different programming approaches: serial implementation, using OpenMP, and using MPI. This document specifically focuses on the MPI approach, although the developed implementation also presents a combined MPI+OpenMP strategy for enhanced performance. It will address topics such as the utilized parallelization approach, the decomposition strategy, synchronization concerns, load balancing, and communication methodology. Additionally, it will present performance results and their respective analysis, alongside a comparison with alternative approaches.

2 - MPI Implementation

2.1 Approach for Parallelization

The parallelization strategy employs MPI directives to enhance the simulation's efficiency. Retaining the original algorithm from the serial implementation, where the next state of each cell is determined based on the current state of its 26 neighbors, distributed parallel constructs are introduced to improve processing speed. This method ensures a consistent basis for comparison with the serial version and facilitates accurate speedup calculations.

Given the nature of the problem, the computation of cell states for each generation remains inherently sequential, as each state depends solely on conditions from the preceding generation. Nevertheless, the task of determining each cell's subsequent state transitions from a sequential to a distributed computing parallel process through the application of MPI directives. In this approach, each machine (or node) in the cluster is assigned a portion of the 3D grid and manages its corresponding segment individually. As memory is distributed across multiple machines, nodes engage in a communication process between the computations of consecutive generations. During this phase, nodes exchange information about the computed cells located at the grid portion boundaries, ensuring each node has the necessary information for the next generation's computation. Additionally, at the end of the computation process of each generation, a `MPI_Reduce` instruction is used to update the `max_cells` and `generation` arrays in the root node, where the maximum number of cells of each species and the generation in which it was reached are stored.

This parallel directive is similarly applied to the `count_cells` function. It enables the concurrent tallying of species populations across the initial grid configuration, with a `MPI_Reduce` instruction used to sum the local counts into the `max_cells` array, thereby optimizing the initial data aggregation phase.

It is also crucial to note that OpenMP directives were utilized inside each node to further enhance parallelization and speed up the simulation time. These instructions were used to divide each node's portion of the grid into t threads, similarly to what was done in the second part of the project. Concerns about the MPI-OpenMP conjugation will be addressed later in the report.

2.2 Decomposition and Load Balancing

The data decomposition strategy involves dividing the 3D grid into smaller segments, which are then assigned to p different processes (one process per node), aiming for the grid to be divided into p equal parts. This section introduces two alternative decomposition approaches that were implemented, with their performance comparison addressed subsequently.

Initially, a *slice decomposition* strategy was employed, dividing the cube along its first dimension, x, into $\frac{N}{p}$ slices, each of size N^2, likewise the strategy used in the OpenMP implementation. However, in a distributed computing environment, where memory is distributed across different nodes, each machine requires data from the two adjacent slices of N^2 elements to compute the next state for its assigned grid segment. To tackle this challenge, two extra slices (ghost cells) are allocated and initialized for each task to store boundary values of adjacent slices from other nodes. Consequently, at the end of every generation, these slices are exchanged between consecutive nodes to compute the next generation efficiently.

Nonetheless, the final version of the project utilized a *checkerboard decomposition*. This method divides the cube ideally into smaller cubes of size $\frac{N^3}{p}$. Figure 1 illustrates a potential division scenario using distributed block decomposition.

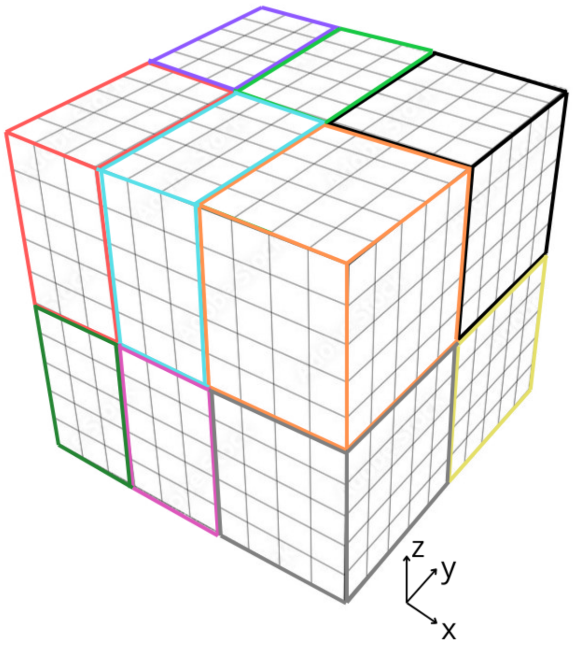

Figure 1: Schematic of 3D Grid Division for $N = 10$ and $p = 12$ tasks

Similar to the slice strategy, the boundary problem is addressed by allocating and initializing ghost cells. However, in this checkerboard version, an extra slice is utilized per cube face, leading to the use of six extra slices. This adjustment results in the allocation of smaller cubes of size $(n+2)^3$, as depicted in Figure 2, with $n = \frac{N}{\sqrt[3]{p}}$ representing the ideal scenario (a perfect cube number of tasks). Throughout the simulation, in each generation, these extra slices are received from adjacent nodes.

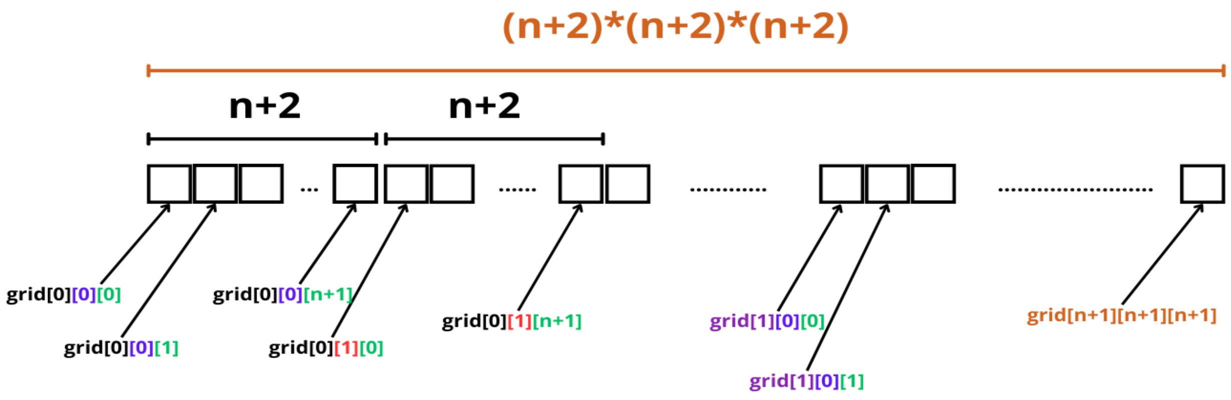

Figure 2: Schematic of 3D grid allocation in the memory of a single node

It is also crucial to mention that each node divides its corresponding sub-cube along its first dimension x and assigns them to the t launched OpenMP threads in an equal fashion. This method, as mentioned before, is the same used in the second part of the project and significantly enhances the computation execution time with reasonable speedups, as the following sections of the report will present.

2.3 Communication Methodology

The communication methodology is categorized into exchanging border data between nodes for their respective grid segments and aggregating cell counts to the root node. The latter uses an MPI reduce operation with $\log(p)$ complexity, efficiently collating simulation results. Border data exchange, however, entails a nuanced approach.

With slice decomposition, nodes exchange two N^2 cell faces per generation using asynchronous communication, which allows the simultaneous computation of border cells and their transmission to corresponding nodes. This reduces the communication overhead by prioritizing border computations and leveraging asynchronous sends.

Checkerboard decomposition introduces serialized communication to manage interactions with six neighbors, necessitating a sequential exchange of plane data along x, then y (already incorporating x's neighbor data), and finally z (with x and y neighbor data included). This sequential strategy ensures all necessary cell states, including corner cells, are efficiently communicated among nodes. To support efficient face exchanges in this schema, the cube's memory allocation was restructured into a continuous block, facilitating the creation of `MPI_Type_vector` for face exchanges and allowing for single send operations per face, as depicted in Figure 2. This arrangement omits borders based on the plane, streamlining the data exchange process.

It is also crucial to mention that nodes are organized into a 3D grid, mirroring the cube division, using `MPI_dims_create` and `MPI_Cart_create` to simplify adjacency identification through `MPI_Cart_shift` and `MPI_Cart_coords`. This grid organization and the serialized plane exchange method in the checkerboard decomposition enhance communication efficiency and algorithmic performance.

2.4 Synchronization Concerns

In a distributed parallel computing implementation, the risk of data races between nodes is inherently null, as memory is not shared. However, without proper synchronization mechanisms, especially during the node communication process, there's a risk that inconsistent states may arise, leading to incorrect or mismatched values across processes. It is crucial to ensure synchronization without compromising the algorithm's efficiency.

As previously discussed, a single generation is processed in parallel by multiple nodes, yet transitioning from one generation to the next necessitates the exchange of information between nodes. This communication is handled by `MPI_Sendrecv`, a blocking operation that completes only once both send and receive actions are finished. Therefore, once `MPI_Sendrecv` returns, the data in the send buffer has been transmitted and can be safely overwritten, and the data in the receive buffer is ready for use. This mechanism ensures that new cell states depend only on the neighbor states from the current generation, already computed and stored in node's memory, eliminating inter-node dependencies in computing the next generation.

Furthermore, `MPI_Reduce` is utilized to ensure data consistency across processes by providing a consistent view of the `local_cells` and `max_cells` arrays. This function gathers data from all processes and applies a reduction operation (sum in this case) to consolidate the data into a single outcome stored in the root process. This operation has a complexity of $\log(p)$, optimizing the aggregation process.

Regarding OpenMP threads within each node, the same precautions described in the second part of the project were applied. The directive `#pragma omp parallel for reduction(+ :  )` functions analogously to `MPI_Reduce`, generating local copies for each task and then aggregating the total value into the node's 'global' array. This ensures efficient parallelization of computation within each node.

2.5 Performance Results and Analysis

This section presents the program's execution times and corresponding speedups for a test scenario provided by the teaching staff. The implementations explored include slices and checkerboard, with the latter evaluated both with and without OpenMP threads. For each MPI process in the slices and the checkerboard without OpenMP, each CPU corresponds to an MPI process. Conversely, with OpenMP, 4 threads are launched, utilizing 4 CPUs per MPI task.

		Decomposition		
		Slices	Checkerboard	
			Without OpenMP	With OpenMP
Total number of CPU's	1	174.3	127.8	96.6
	2	87.2 (1.999)	64 (1.997)	-
	4	43.6 (3.998)	32.1 (3.981)	-
	8	21.8 (7.995)	16.1 (7.938)	-
	16	10.9 (15.991)	8.1 (15.778)	6.9 (14.000)
	32	5.5 (31.691)	4.1 (31.171)	3.5 (27.600)
	64	2.8 (62.250)	2.1 (60.857)	1.8 (53.667)

Table 1: Execution time in seconds (and speedups) for `./life3d-mpi 3 1024 .4 100` with the different implemented versions; **4 threads** were used when testing the versions with OpenMP, excluding the first one where **OMP_NUM_THREADS=1**.

Table 1 indicates significant speed enhancements up to 64 CPUs for the slices implementation, exceeding 97% efficiency. The checkerboard implementation also shows notable scalability with speedups exceeding 83% and 95% efficiency (with and without OpenMP threads respectively).

Although the slices version exhibits slightly better speedups, the checkerboard methodology yields lower execution times. This disparity suggests that for larger 3D grids or more generations, the checkerboard approach's advantages could become more pronounced due to its superior scalability compared to other decomposition methods.

Furthermore, employing OpenMP results in reduced overhead from thread creation and management compared to MPI processes, as threads are more lightweight. Additionally, communication overhead is minimized since larger memory chunks are shared within the same machine rather than being distributed across multiple independent memory spaces. This key factor contributes to lower execution times with OpenMP threads versus MPI processes for the same total number of CPUs used concurrently.

3 - Conclusions

This report aimed to optimize the 3D Game of Life simulation by applying parallel computing strategies, focusing primarily on MPI for enhancing performance. The main approach involved data decomposition and load balancing, dividing the computational grid into equal segments for processing by multiple machines. Communication protocols were carefully implemented to avoid data inconsistencies, ensuring the integrity of the simulation's results. The checkerboard decomposition approach was employed to distribute work among nodes effectively, minimizing communication overhead and maximizing resource utilization. Empirical results, as demonstrated through execution times and speedups, indicate substantial performance improvements. Overall, the objectives were successfully met, showcasing the efficacy of parallel processing techniques in improving computational tasks.